

# Compilado preguntas rios (y otros)

Logico:

- **Hablar de paradigma logico, pregunta inicial porque le dije que logico me llamo la atencion porque fue totalmente nuevo. (También mencionar que está muy loco lo de la reversibilidad)**

*Resumen rápido:*

*Queremos usar lógica para programar, se basa en usar hechos (algo así como axiomas) y reglas de inferencia, a través de esto se buscan soluciones.*

Proposicional:

-Arranca desde la premisa que si una fórmula  $A$  es tautología (vale para toda valuación) entonces  $\neg A$  es una contradicción. Así que la negamos y seguimos.

- Le sumamos que  $A = (P \vee \neg Q) \wedge (J \vee Q)$  es equivalente a  $A' = (P \vee \neg Q) \wedge (J \vee Q) \wedge (P \vee J)$  (base de la regla de resolución)

- Entonces reescribimos la fórmula en forma normal conjuntiva, pasa por una serie de transformaciones que son equivalentes:

1. Las negaciones se aplican a literales
2. Es una conjunción de disyunción de literales  $= (C1 \vee C2) \wedge (C3 \vee C4)$
3. Se escribe en notación conjuntista  $(C1 \vee C2) \wedge (C3 \vee C4) = \{ \{C1, C2\}, \{C3 \vee C4\} \}$

*Con todo esto tengo algo equivalente a  $\neg A$  y quiero probar que es insatisfacible usando la regla de resolución, que preserva insatisfacibilidad.*

$\{A1, \dots, Am, Q\} \quad \{B1, \dots, Bn, \neg Q\}$

-----

$\{A1, \dots, Am, B1, \dots, Bn\} \rightarrow$  Resolvente

*Vamos agregando las resolventes al conjunto buscando llegar a una contradicción, que seria la resolvente vacia, cuando nos queda algo del estilo*

$\{\dots \neg Q\}, \{Q\} \rightarrow$  Paso de resolución  $\{ \square \}$

*Entonces  $\neg A$  es una contradicción, por lo que  $A$  era tautología.*

*Si no encuentro contradicción,  $A$  no era tautología*

**Rodri.K: Esta es una diferencia remarcable con primer orden, donde debido a skolemizar no puedo hacer las mismas afirmaciones sobre validez logica.**

Primer Orden:

- Aparecen los cuantificadores, funciones, predicados y formulas, las variables se pueden ligar, etc

- Una fórmula es satisfacible si existe una estructura  $M$  en la que es satisfacible (o sea, si tiene una asignación en esa estructura tal que la hace verdadera) y es válida en  $M$  si es satisfacible para toda asignación, o sea  **$A$  ES VALIDA SI ES VALIDA PARA TODA ESTRUCTURA  $M$ , Y  $A$  ES VÁLIDA  $\Leftrightarrow \neg A$  INSATISFACIBLE**

Pero aparece el problemita del teorema de Church, no existe un algoritmo que pueda determinar si una fórmula de primer orden es válida, por eso lo que hagamos son mecanismos de semidecisión, si una sentencia es insatisfacible encontramos refutación, pero si es satisfacible puede ser que no se detenga.

Mismo procedimiento que en proposicional, pero hay que toquetear un poco antes, negamos la expresión y la escribimos en forma clausal, siguiendo los pasos:

- 1) Remover implicación
- 2) Pasar a forma normal negada (todos los  $\neg$  tiene que ir con los átomos)
- 3) Opcional, forma normal prenexa, mandar los cuantificadores adelante.
- 4) Skolemizar, preserva satisfacibilidad pero no validez, cada variable alcanzada por un  $\exists$  se tiene que reemplazar: Si no hay variables libres en el término es por una constante y si hay variables libres es por una función que toma como parámetros esas variables libres. Luego se eliminan los  $\exists$
- 5) Pasar a forma normal conjuntiva (conjunción de disyunción de literales  $(P \vee Q) \wedge (J \vee R)$ )
- 6) Distribuir cuantificadores universales  $\rightarrow$  escribir en notación de conjuntos

Aplicamos método de resolución:

- Si hallamos una refutación,  $\neg A$  es insatisfacible  $\Rightarrow A$  es tautología
- Si no hallamos refutación,  $\neg A$  es satisfacible  $\Rightarrow A$  no es tautología

Regla de resolución en PO, podemos agarrar de más de un literal, y tenemos que usar MGU porque puede pasar que  $\{p(a)\}\{\neg p(x)\}$

$\{B_1, \dots, B_k, A_1, \dots, A_m\} \quad \{\neg D_1, \dots, \neg D_j, C_1, \dots, C_n\}$

-----  
 $\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\}) \rightarrow$  Resolvente

Donde  $\sigma$  es el MGU de  $\{B_1, \dots, B_k, \neg D_1, \dots, \neg D_j\}$

Notar que  $\sigma(B_1) = \dots = \sigma(B_k) = \sigma(\neg D_1) = \dots = \sigma(\neg D_j)$

Regla de resolución binaria

$\{B, A_1, \dots, A_m\} \quad \{\neg D, C_1, \dots, C_n\}$

-----  
 $\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\})$

Es incompleta,

$\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}\}$

$\{\{P(x), P(y)\},$   
 $\{\neg P(v), \neg P(w)\}\}$

$(x < -a, v < -a) \text{ , 1 y 2}$

$\{P(y), \neg P(w)\}$

$(x < -a, w < -a) \text{ 1 y 3}$

$\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(y), \neg P(w)\}, \{P(x), P(y)\}\}$

se puede recuperar la completitud factorizando

$\{\{P(x), P(y)\},$   
 $\{\neg P(v), \neg P(w)\}, \{P(z), \neg P(t)\}$

Y ahora si puedo llegar a la contradicción

-----

Con todo esto se tiene un método de resolución para PO que es **completo**, pero en la práctica es muy costoso buscar soluciones, el espacio de búsqueda puede ser enorme y hay un alto grado de no determinismo. Hay que empezar a poner reglas (**búsqueda y selección**) para reducir el espacio de búsqueda, sería deseable no renunciar a la completitud del método.

Resolución lineal: Elige de a dos y la resolvente resultante lo usa en otro conjunto de literales.

Preserva completitud y es más rápido, pero sigue siendo altamente no determinístico y no especifica criterio de selección. Aparecen las **CLAUSULAS DE HORN**

Clausulas de HORN: Es una disyunción de literales con A LO SUMO un literal positivo

**NO TODA FÓRMULA de PO PUEDE SER EXPRESADA CON ESTO, PERO ALCANZA PARA LOS PROGRAMAS, ES COMPLETO EN ESTA SUBCLASE.**

Con esto podemos usar **RESOLUCIÓN SLD**.

-  $S = P \cup \{G\}$  un conjunto de clausulas de Horn con nombres de variables disjuntos con **exactamente un literal positivo y un goal**

Son las clausulas de entrada: (\*)

-  $P$  es un conjunto tal que la disyunción de literales tiene **EXACTAMENTE** un literal positivo

Es el programa o base de conocimiento.

-  $G = (\text{Goal})$  es una cláusula negativa,

**Se arranca desde el goal y se aplica resolución. SLD no especifica ni regla de selección ni de búsqueda**

- SLD es Correcto (Un conjunto de clausulas de Horn (definidas en \*) con refutación SLD, entonces esas clausulas son insatisfacibles)
- SLD es completo (Dado un conjunto de clausulas de Horn (definidas en \*) es insatisfacible, SLD encuentra refutación)

Prolog usa SLD, pero agrega los criterios de búsqueda (arriba para abajo) y selección (izquierda a derecha). Esta estrategia define un árbol que se recorre DFS.  
 La implementación de prolog no es completa, porque puede no encontrar una refutación aunque exista. Puedo asegurar corrección: Si encuentro una refutación con Prolog (SLD + Reglas de selección y búsqueda + backtracking) el programa + el goal (consulta negada) es insatisfactible

=====

**- Como es el mecanismo de resolución en general (escribirlo)**

Listo

=====

**- Que es el MGU:**

Es el unificador mas general, es la solución al conjunto de ecuaciones de unificación del estilo  $\{\sigma_1 \triangleq \sigma_1', \sigma_2 \triangleq \sigma_2', \dots\}$ , y además es la mas general porque cualquier otra solución  $T$  que se encuentre es una instanciación de este MGU

- $\{V \times \text{Nat} \rightarrow \text{Nat} \triangleq u \rightarrow \text{Nat}\}$  tiene como solución
  - $\{V \times \text{Nat}/u\}$  es MGU del conjunto
  - $\{\text{Bool}/V, \text{Bool} \times \text{Nat}/u\}$  es solución pero no es MGU porque es instanciación del anterior!

=====

**- Como se calcula el mgu**

Con el algoritmo de Martelli Montanari, consiste en reglas de simplificación que simplifican pares de tipos a unificar, de la pinta  $\sigma_1 \triangleq \sigma_2$ . Cabe aclarar que es un algoritmo no determinístico.

1. Descomposición
2. Eliminación de par trivial
3. Swap (variables de tipo a la izquierda)
4. Eliminación de variable (si  $s \text{ /notbelongs } FV(\sigma)$ ), aplicación de la sustitución
5. Colisión (dos tipos basicos no unificables)
6. Occur Check, Es decir, se generaria una especie de recursión finita, reemplazando  $s$  por  $\sigma$ , pero estando  $s$  adentro de  $\sigma$

=====

**Lambda:**

**- Dar las reglas de semantica operacional con aplicacion (M N)**

$$\begin{array}{l}
 M_1 \rightarrow M'_1 \\
 \hline \text{E-App1} \\
 M_1 M_2 \rightarrow M'_1 M_2 \\
 \\
 M_2 \rightarrow M'_2 \\
 \hline \text{E-App2} \\
 V M_2 \rightarrow V M'_2 \\
 \\
 \hline \text{E-AppAbs} \\
 (\lambda x:\sigma.M) V \rightarrow M\{x \leftarrow V\}
 \end{array}$$

### =====

**- Por que hay que tener cuidado con las variables libres en aplicación y como se soluciona**

Porque en la sustitución puede que se termine capturando la ocurrencia de las variables libres,

- ej:  $(\lambda z:\sigma.x)\{x \leftarrow z\} = \lambda z:\sigma.z$ .
  - La función constante  $x$  se transforma en la función identidad
  - Se soluciona renombrando el nombre de la variable ligada, ya que no son relevantes. Generando una nueva fórmula  $\alpha$ -equivalente.

### =====

**- Como se extiende lambda con referencias (solo dar las reglas de tipado). De paso hable de unit y para que se pone (toque el tema por  $a := b$ )**

*Quieres emular lenguajes imperativos, cambios de estados, agregas 3 operaciones;*

*Asignación, lectura y asignación de memoria*

1. *Ref M  $\rightarrow$  Genera una referencia al término M*
2. *M := N  $\rightarrow$  Guarda N en M (M tiene que ser una referencia)*
3. *N!  $\rightarrow$  Lee el valor almacenado en N*

- *Terminos:: ... | !M | Ref M | M := N | I*
- *Valores:: ... | unit | I*
- *Tipos:: ... | Unit | Ref  $\sigma$  | I (No lo quito pero lo marco, I no es un tipo, es un valor con tipo Ref  $\sigma$ )*

*En el tema de tipado me interesan los efectos, entonces aparece Unit para tipar algo del estilo M := N, es una especie de Void.*

- *Las referencias tienen un tipo, y hay que llevar cuenta de esto*
- *Además tener una memoria (ahora eso puede cambiar, I1 puede tener una cosa y después otra), aparece el store  $\mu$  que es una función que toma una dirección y devuelve el elemento almacenado ahí,*
- *hace falta un contexto de tipado  $\Sigma$  para direcciones de memoria, que es una función de direcciones de memoria a tipos,*

- las reglas de tipado son:

$$\begin{array}{c}
 \frac{\Gamma \mid \Sigma > M : \sigma}{\Gamma \mid \Sigma > \text{Ref } M : \text{Ref } \sigma} \quad T\_ref \\
 \\
 \frac{\Gamma \mid \Sigma > M : \text{Ref } \sigma}{\Gamma \mid \Sigma > !M : \sigma} \quad T\_deref \\
 \\
 \frac{\Gamma \mid \Sigma > M : \text{Ref } \sigma \quad \Gamma \mid \Sigma > N : \sigma}{\Gamma \mid \Sigma > M := N : \text{Unit}} \quad T\_assign \\
 \\
 \frac{\Sigma(l) = \sigma}{\Gamma \mid \Sigma > l : \text{Ref } \sigma} \quad T\_loc
 \end{array}$$

Nota: Al agregar referencias se pierde progreso y preservación.

- Progreso: Dado  $M$  Cerrado y bien tipado,
  - $M$  es un valor,
  - o existe un  $M' / M \rightarrow M'$
- Preservación :  $\Gamma > M : \sigma$  ,  $M \rightarrow N$ , entonces  $\Gamma > N : \sigma$

Para arreglarlo hay que reformular

- Hay que compatibilizar los tipos de memoria y el contexto de tipado. Hay que tipar la memoria.

## =====

### - Que es currificacion y uncurry (codearlo) y para que sirve, map (+3)

- Currificar es el proceso de transformar una funcion que recibe parámetros en una función de alto orden (intermedia) que recibe un parámetro menos y termina el trabajo. Permite mayor expresividad, modularidad en el código y poder aplicar evaluación parcial.
- Curry  $f = g$  where  $g \ x \ y = f(x,y)$
- Uncurry  $f = g$  where  $g(x,y) = f \ x \ y$

## =====

### Objetos:

- hablar sobre method dispatch (en realidad me pidio que le hable de objetos y agarre por ahi con super y self). Hable de que se resuelve en compilación o ejecución y como buscan los metodos para cada mensaje

Objetos: todo programa es una simulación, y cada entidad simulada se representa con objetos, que tienen las características y capacidades del mundo real que nos interesa modelar. Cada objeto oculta sus datos (sólo accesibles desde las operaciones del objeto) y ofrece un protocolo de mensajes para interactuar con los otros objetos.

Clases: modelan conceptos abstractos del dominio del problema a trabajar. Definen el comportamiento y la forma de un conjunto de objetos. Todo objeto es instancia de alguna clase. Las clases son, a su vez, objetos.

Method dispatch: al recibir un mensaje, es el proceso que lo relaciona con el método (código que define el comportamiento del objeto) a ejecutar. Los métodos tienen nombre (selector), parámetros y cuerpo. El method dispatch puede ser en tiempo de compilación (MD estático), o en tiempo de ejecución (MD dinámico).

Hay dos pseudovariables, self y super.

- Self referencia al receptor del mensaje que activó la evaluación del método, y se liga cuando comienza dicha evaluación.
- Super referencia al mecanismo del method lookup; se hace desde el padre del método en ejecución.

=====

**Logico:**

**-cual es el mecanismo que le da sentido (o algo así la pregunta, pero la respuesta era resolución) y me tiro que hable más de eso. Luego me pidió que escriba la regla en el pizarro y la escribí de proposicional y de primer orden, y me pregunto cosas de esas dos reglas, que es unificar, y me tiro de la regla de resolución binaria, que problema tenía**

Esta todo más arriba,

=====

**-me pregunto que es lo bueno y malo del paradigma lógico.**

*Bueno, reversibilidad: que se pueden preguntar distintos tipos de resultados sobre una misma cosa. Por ejemplo, en el caso de un predicado de suma con 3 parámetros, podemos preguntar tanto por verdadero/falso ( $X + Y$  son iguales a  $Z$ ?), como por los valores específicos (dados  $X$  e  $Y$ , qué  $Z$  suman? qué  $Y$  le puedo sumar a  $X$  para que de  $Z$ ? Dado  $Z$ , qué  $X$  e  $Y$  suman ese valor? etc.).*

*Es decir, no hay parámetros de entrada ni de salida, lo que se define es una relación entre los elementos.*

*Malo: Prolog no es completo, en resolución van a haber soluciones que se puedan alcanzar, que por los mecanismos que utiliza prolog de búsqueda y selección, se están omitiendo, ie: un mismo programa pero variando la regla de selección desde la derecha, puede generar diferentes resultados. Ejemplo:*

*$p(X)$ :-  $p(X)$ .*

*$p(1)$ .*

=====

**Lambda:**

**Me tiro que hable de inferencia de tipos, cual es el problema y que lo enuncie en el pizarrón. No lo sabía pero lo chamuye, y después me hizo preguntas de lo que escribí y si siempre funciona ese algoritmo (puede fallar si no tipa).**

*La inferencia consiste en transformar términos sin información de tipos o con información parcial, en términos tipables. De ahí que se tiene que inferir la información de tipos faltante.*

- *El programador puede obviar declaraciones*
- *Evita la sobrecarga de tener que declarar y manipular todos los tipos*
- *No desmejora la performance, ya que la inferencia se hace en tiempo de compilación*

*Hay dos cosas, **inferencia** de tipos y **chequeo**.*

- *Inferencia, dado un término  $U$  sin anotaciones de tipo encontrar un  $M$ , tal que*
  - $\Gamma \vdash M:\sigma$  para algún  $\Gamma$  y  $\sigma$
  - $\text{erase}(M) = U$
- *Chequeo, dado un término  $M$  con anotaciones de tipo, determinar si existen  $\Gamma$  y  $\sigma$  /*
  - $\Gamma \vdash M:\sigma$  es derivable

*El algoritmo para inferencia  $W(\cdot)$  se define por recursión sobre la estructura del término  $U$  y usa el algoritmo de unificación de Martelli Montanari. Puede fallar si, por ejemplo, falla la unificación.*

$W(0) = \{\} \triangleright 0:\text{Nat}$

$W(\text{true}) = \{\} \triangleright \text{true}:\text{bool}$

$W(\text{false}) = \{\} \triangleright \text{false}:\text{bool}$

$W(x) = \{x:s\} \Rightarrow X:s$ , con  $s$  fresca

**Succ, pred**

$W(U) = \Gamma \triangleright M:\tau$

Sea  $S = \text{MGU}\{\tau = \text{Nat}\}$

$W(\text{succ}(U)) = S\Gamma \triangleright SSucc(M):\text{Nat}$

*Y así, es una paja llenarlos, creo que el que más vale es el de la abstracción*

$W(U) = \Gamma \triangleright M:\rho$

- *Si  $\Gamma$  tiene información de tipos para  $x$ , hay que sacarla del contexto y tiparlo*

$W(\lambda x.U) = \Gamma / \{x:\tau\} \triangleright \lambda x:\tau. M : \tau \rightarrow \rho$

- *Si no tiene, hay que agarrar una variable Fresca  $s$*

$W(\lambda x.U) = \Gamma \triangleright \lambda x:s. M : s \rightarrow \rho$

**Conclusiones del algoritmo:**

- *Los llamados recursivos devuelven un contexto, un término anotado y un tipo.*



- Cuando la regla tiene tipos iguales => unifica
- Si hay contextos repetidos en las premisas => unificarlos
- Cuando la regla liga variables:
  - Obtener su tipo del contexto obtenido recursivamente
  - Si no están, variable fresca
  - Sacarlas del contexto resultado, ya que dejaron de ser variables libres
- Decorar términos
- Si la regla tiene restricciones adicionales, se incorporan como casos posible de falla.

=====

**Melanie Sclar: Toma teórico, no me hizo hacer ningún ejercicio. Fue oral, y me preguntó cosas de Haskell conceptuales (mencionar las características del lenguaje que vimos en la materia).**

*Lenguaje funcional puro: lenguaje de expresiones con transparencia referencial (el valor de una expresión depende sólo de los elementos que la constituyen), y funciones como valores (que pueden ser a su vez pasadas como parámetros de otras funciones -> alto orden), cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de iguales por iguales, definidos por ecuaciones orientadas. La información se "transforma", no hay interacción con el medio.*

=====

**Después me preguntó sobre la extensión del cálculo lambda para agregar memoria (cuáles son los valores, términos, reglas).**

Esto está antes.

=====

**Finalmente le conté resolución SLD y por qué se puede aplicar a Prolog y de objetos no me preguntó nada.**

Esto está antes.

=====

**Juan Manuel Pérez: Aprendete los conceptos de Currificación y Polimorfismo paramétrico.**

*Currificación: antes*

*Polimorfismo paramétrico: es una característica del sistema de tipos. Dada una expresión, que puede ser tipada de infinitas maneras (por ejemplo, la función identidad), el sistema le asigna*

*un tipo más general que todos ellos (usando variable de tipos), de manera que en cada uso, pueda transformarse en un tipo particular. Es paramétrico*

=====

**Cálculo Lambda con todas sus extensiones (memoria, let, etc). Los toma casi siempre estos temas.**

*Paja, no voy a escribir todo eso*

=====

**2- Qué es currificación? Escribir la función curry**

*Esto está antes.*

=====

**3- Cuando hacemos sustitución en lambda cálculo hay un caso que hay que prestar atención.Cuál es y qué hay que tener en cuenta?**

Esto está antes. La captura de variables

=====

**4- En que se basa la programación lógica para funcionar? Cómo es el método para lógica proposicional y para LPO? En qué se diferencian?**

Está antes

=====

**5- Algo que te haya llamado la atención de programación lógica?**

Está antes

=====

**Me tomo curry,**

Esto ya está antes.

=====

**Lambda con memoria**

Esto ya está antes.

## Subtipado de funciones:

El sistema de tipado es muy estricto, y puede descartar programas válidos:

$(\lambda x : \{a : \text{Nat}\}.x.a)\{a = 1, b = \text{true}\}$

Se introduce principio de sustituibilidad:

$\sigma <: \tau$

“En todo contexto donde se espera  $\tau$  puedo usar una de tipo  $\sigma$  sin que esto genere un error”

$\Gamma \vdash M : \sigma \quad \sigma <: \tau$

-----  $t_{\text{sub}}$

$\Gamma \vdash M : \tau$

Para el subtipado de funciones, se usa  $s_{\text{func}}$ .

Uno tiene una función  $F$  que va de  $\sigma \rightarrow \tau$  y la quiere subtipar por otra  $G$  que va de  $\sigma' \rightarrow \tau'$ .

Se piensa en el dominio de  $F$  y de  $G$ ,  $G$  tiene que poder responder todo lo que puede recibir  $F$ , o sea que  $\text{Dom}(G) \supseteq \text{Dom}(F)$ .  $\sigma' \supset \sigma$ .

Con la imagen es al revés, vos estas devolviendo algo que devolveria  $F$ , así que no te podes “pasar”

$\tau' < \tau$ . O sea que es contravariante en el dominio, y covariante en la image

$\sigma <: \sigma' \quad \tau' <: \tau$

-----  $s_{\text{func}}$

$\sigma' \rightarrow \tau' <: \sigma \rightarrow \tau$

Se hace del siguiente modo: toda aplicación de  $f$  será a argumentos de tipo  $\sigma$ , que se coercionan a algo de tipo  $\sigma'$ . Se aplica la función  $G$ , cuyo tipo real es  $\sigma' \rightarrow \tau'$ . Finalmente, el resultado se coercion a  $\tau$ , el tipo del resultado que el programa  $P$  estaba esperando.

## Resolucion en logica proposicional y lpo

Esto ya está antes.

También qué es mgu, para qué se usa en inferencia de tipos, que es a-equivalencia.

MGU  $\rightarrow$  ya está.

*Dos términos son alpha-equivalentes cuando sólo difieren en el nombre de sus variables ligadas.*

*Inferencia de tipos: el MGU se usa en los distintos tipos del algoritmo de inferencia de tipos. En otra pregunta está explicado cómo calcular el MGU, y está detallado además el algoritmo de inferencia de tipos.*

=====

**Ademas le hable de not y cut**

*Cut: sirve para hacer más eficiente la búsqueda en la resolución de Prolog, ya que poda ramas del árbol de búsqueda. Lo que se haya unificado hasta ese momento, es lo que vale, y si se tiene que hacer backtracking sobre lo ya unificado, falla. No da más soluciones del goal padre, de cualquier goal que ocurre a la izquierda del corte en la cláusula que lo contiene, y todos los objetivos intermedios que se ejecutaron durante la ejecución de los goals precedentes.*

*Not: no es la negación lógica, sino que es la negación por falla. No significa que se cumple no  $P$  (esto es indecidible), sino que  $P$  no puede ser probado. Se define de la siguiente manera:*

*$\text{not}(P) \text{ :- call}(P), !, \text{fail}.$   
 $\text{not}(P).$*

*Esto significa que si  $P$  tiene éxito, entonces ahí se corta el backtracking (no se siguen buscando definiciones), y el llamado  $\text{not}(P)$  falla. Si  $P$  falla, entonces  $\text{not}(P)$  tiene éxito (porque entra a la segunda definición).*